

AD-A199 426

THE FILE

2

AVF Control Number: AVF-VSR-017
SZT-AVF-017

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 88022911.09044
SYSTEM KG
SYSTEM Ada Cross Compiler VAX/VMS x MC68020/OS-9
Version 1.61
VAX 8530, KVS M68K

Completion of On-Site Testing:
88-02-29

Prepared By:
IABG m.b.H., Dept SZT
Einsteinstrasse 20
8012 Ottobrunn
Federal Republic of Germany

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington, D.C. 20301-3081

DTIC
ELECTE
S AUG 31 1988 D
H

Ada is a registered trademark of the United States Government
(Ada Joint Program Office).

DISTRIBUTION STATEMENT A

Approved for public release;
Distribution Unlimited

88 8 31 027

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		REAL INSTRUCTIONS RECEIVED DATE
1. REPORT NUMBER	12. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Ada Compiler Validation Summary Report: SYSTEAM KG, SYSTEAM Ada Compiler VAX/VMS x MC68020/OS-9 Version 1.61, VAX 8530 (Host) and KWS EB 68/20 (Target).		5. TYPE OF REPORT & PERIOD COVERED 29 Feb 1989 to 29 Feb 1989
7. AUTHOR(s) IABG, Ottobrunn, Federal Republic of Germany		6. PERFORMING ORG. REPORT NUMBER
9. PERFORMING ORGANIZATION AND ADDRESS IABG, Ottobrunn, Federal Republic of Germany		8. CONTRACT OR GRANT NUMBER(s)
11. CONTROLLING OFFICE NAME AND ADDRESS Ada Joint Program Office United States Department of Defense Washington, DC 20301-3081		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) IABG, Ottobrunn, Federal Republic of Germany		12. REPORT DATE 29 February 1988
		13. NUMBER OF PAGES 52 p.
		15. SECURITY CLASS (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20. If different from Report) UNCLASSIFIED		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Ada Programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, ACVC, Validation Testing, Ada Validation Office, AVO, Ada Validation Facility, AVF, ANSI/MIL-STD- 1815A, Ada Joint Program Office, AJPO		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) SYSTEAM Ada Compiler VAX/VMS x MC68020/OS-9 Version 1.61, SYSTEAM KG, IABG, VAX 8530 under VMS, Version 4.6 (Host) and KWS EB 68/20 CPU3 under OS-9, Version 2.0 (Target), ACVC 1.9.		

DD FORM 1473
1 JAN 73EDITION OF 1 NOV 68 IS OBSOLETE
S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Ada Compiler Validation Summary Report:

Compiler Name: SYSTEAM Ada Cross Compiler VAX/VMS x MC68020/OS-9
Compiler Version: Version 1.61

Certificate Number: 880229I1.09044

Host:

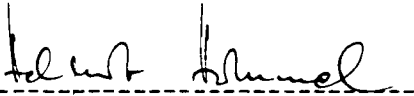
VAX 8530 under
VMS,
Version 4.6

Target:

KWS EB 68/20 CPU3 under
OS-9,
Version 2.0

Testing Completed 88-02-29 Using ACVC 1.9

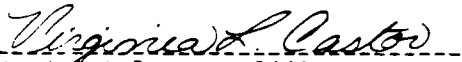
This report has been reviewed and is approved.



IABG m.b.H., Dept SZT
Dr. H. Hummel
Einsteinstrasse 20
8012 Ottobrunn
Federal Republic of Germany



Ada Validation Organization
Dr. John F. Kramer
Institute for Defense Analyses
Alexandria VA 22311



Ada Joint Program Office
Virginia L. Castor
Director
Department of Defense
Washington DC 20301

CONTENTS

CHAPTER 1	INTRODUCTION	
1.1	PURPOSE OF THIS VALIDATION SUMMARY REPORT	1-2
1.2	USE OF THIS VALIDATION SUMMARY REPORT	1-2
1.3	REFERENCES	1-3
1.4	DEFINITION OF TERMS	1-4
1.5	ACVC TEST CLASSES	1-5
CHAPTER 2	CONFIGURATION INFORMATION	
2.1	CONFIGURATION TESTED	2-1
2.2	IMPLEMENTATION CHARACTERISTICS	2-1
CHAPTER 3	TEST INFORMATION	
3.1	TEST RESULTS	3-1
3.2	SUMMARY OF TEST RESULTS BY CLASS	3-1
3.3	SUMMARY OF TEST RESULTS BY CHAPTER	3-2
3.4	WITHDRAWN TESTS	3-2
3.5	INAPPLICABLE TESTS	3-2
3.6	TEST, PROCESSING, AND EVALUATION MODIFICATIONS	3-4
3.7	ADDITIONAL TESTING INFORMATION	3-4
3.7.1	Prevalidation	3-5
3.7.2	Test Method	3-5
3.7.3	Test Site	3-5
APPENDIX A	DECLARATION OF CONFORMANCE	
APPENDIX B	APPENDIX F OF THE Ada STANDARD	
APPENDIX C	TEST PARAMETERS	
APPENDIX D	WITHDRAWN TESTS	



Accession For	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

CHAPTER 1

INTRODUCTION

This Validation Summary Report (VSR) describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability (ACVC). An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation dependent but permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

INTRODUCTION

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

- . To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard
- . To attempt to identify any language constructs not supported by the compiler but required by the Ada Standard
- . To determine that the implementation-dependent behavior is allowed by the Ada Standard

Testing of this compiler was conducted under the direction of the AVF according to procedures established by the Ada Joint Program Office and administered by the Ada Validation Organization (AVO). On-site testing was completed 88-02-29 at SYSTEAM KG at Karlsruhe.

1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

Ada Information Clearinghouse
Ada Joint Program Office
OUSDRE
The Pentagon, Rm 3D-139 (Fern Street)
Washington DC 20301-3081

or from:

IABG m.b.H., Dept SZT
Einsteinstrasse 20
8012 Ottobrunn
Federal Republic of Germany

INTRODUCTION

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.3 REFERENCES

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
2. Ada Compiler Validation Procedures and Guidelines, Ada Joint Program Office, 1 January 1987.
3. Ada Compiler Validation Capability Implementers' Guide, SofTech, Inc., December 1986.
4. Ada Compiler Validation Capability User's Guide, December 1986.

INTRODUCTION

1.4 DEFINITION OF TERMS

ACVC	The Ada Compiler Validation Capability. The set of Ada programs that tests the conformity of an Ada compiler to the Ada programming language.
Ada Commentary	An Ada Commentary contains all information relevant to the point addressed by a comment on the Ada Standard. These comments are given a unique identification number having the form AI-ddddd.
Ada Standard	ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
Applicant	The agency requesting validation.
AVF	The Ada Validation Facility. The AVF is responsible for conducting compiler validations according to procedures contained in the <u>Ada Compiler Validation Procedures and Guidelines</u> .
AVO	The Ada Validation Organization. The AVO has oversight authority over all AVF practices for the purpose of maintaining a uniform process for validation of Ada compilers. The AVO provides administrative and technical support for Ada validations to ensure consistent practices.
Compiler	A processor for the Ada language. In the context of this report, a compiler is any language processor, including cross-compilers, translators, and interpreters.
Failed test	An ACVC test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.
Host	The computer on which the compiler resides.
Inapplicable test	An ACVC test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.
Passed test	An ACVC test for which a compiler generates the expected result.
Target	The computer for which a compiler generates code.
Test	A program that checks a compiler's conformity regarding a particular feature or a combination of features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.
Withdrawn test	An ACVC test found to be incorrect and not used to check conformity to the Ada Standard. A test may be incorrect

because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce compilation or link errors.

Class A tests check that legal Ada programs can be successfully compiled and executed. There are no explicit program components in a Class A test to check semantics. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that every syntax or semantic error in the test is detected. A Class B test is passed if every illegal construct that it contains is detected by the compiler.

Class C tests check that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers permitted in a compilation or the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

INTRODUCTION

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated.

Two library units, the package REPORT and the procedure CHECK_FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of the tests in the ACVC follow conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and place features that may not be supported by all implementations in separate tests. However, some tests contain values that require the test to be customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of this validation are given in Appendix D.

CHAPTER 2
CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

Compiler: SYSTEAM Ada Cross Compiler VAX/VMS x MC68020/OS-9,
Version 1.61

ACVC Version: 1.9

Certificate Number: 86022911.09044

Host Computer:

Machine:	VAX 8530
Operating System:	VMS Version 4.6
Memory Size:	32 MB

Target Computer:

Machine:	KWS EE 68/20 CPU3
Operating System:	OS-9 Version 2.0
Memory Size:	2.0 MB

Communications Network: V24 connection

CONFIGURATION INFORMATION

2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of a compiler in those areas of the Ada Standard that permit implementations to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. The tests demonstrate the following characteristics:

- Capacities.

The compiler correctly processes tests containing loop statements nested to 65 levels, block statements nested to 65 levels, and recursive procedures separately compiled as subunits nested to 17 levels. It correctly processes a compilation containing 723 variables in the same declarative part. (See tests D55A03A..H (8 tests), D56001B, D64005E..G (3 tests), and D29002K.)

- Universal integer calculations.

An implementation is allowed to reject universal integer calculations having values that exceed `SYSTEM.MAX_INT`. This implementation processes 64 bit integer calculations. (See tests D4A002A, D4A002B, I4A004A, and D4A004B.)

- Predefined types.

This implementation supports the additional predefined types `SHORT_INTEGER`, `SHORT_FLOAT`, and `LONG_FLOAT` in the package `STANDARD`. (See tests B86001C and B86001D.)

- Based literals.

An implementation is allowed to reject a based literal with a value exceeding `SYSTEM.MAX_INT` during compilation, or it may raise `NUMERIC_ERROR` or `CONSTRAINT_ERROR` during execution. This implementation raises `NUMERIC_ERROR` during execution. (See test E24101A.)

- Expression evaluation.

Apparently no default initialization expressions for record components are evaluated before any value is checked to belong to a component's subtype. (See test C32117A.)

Assignments for subtypes are performed with the same precision as the base type. (See test C35712B.)

CONFIGURATION INFORMATION

This implementation uses no extra bits for extra precision. This implementation uses all extra bits for extra range. (See test C35903A.)

Apparently NUMERIC_ERROR is raised when an integer literal operand in a comparison or membership test is outside the range of the base type. (See test C45232A.)

Apparently NUMERIC_ERROR is raised when a literal operand in a fixed-point comparison or membership test is outside the range of the base type. (See test C45252A.)

Apparently underflow is gradual. (See tests C45524A..Z.)

Rounding.

The method used for rounding to integer is apparently round to even. (See tests C46012A..Z.)

The method used for rounding to longest integer is apparently round to even. (See tests C46012A..Z.)

The method used for rounding to integer in static universal real expressions is apparently round away from zero. (See test C4A014A.)

CONFIGURATION INFORMATION

Array types.

An implementation is allowed to raise `NUMERIC_ERROR` or `CONSTRAINT_ERROR` for an array having a `'LENGTH` that exceeds `STANDARD.INTEGER'LAST` and/or `SYSTEM.MAX_INT`. For this implementation:

Declaration of an array type or subtype declaration with more than `SYSTEM.MAX_INT` components raises no exception. (See test C36003A.)

`NUMERIC_ERROR` is raised when `'LENGTH` is applied to an array type with `INTEGER'LAST + 2` components. (See test C36202A.)

`NUMERIC_ERROR` is raised when `'LENGTH` is applied to an array type with `SYSTEM.MAX_INT + 2` components. (See test C36202B.)

A packed `BOOLEAN` array having a `'LENGTH` exceeding `INTEGER'LAST` raises `NUMERIC_ERROR` when the array type is declared. (See test C52103X.)

A packed two-dimensional `BOOLEAN` array with more than `INTEGER'LAST` components raises `NUMERIC_ERROR` when the array type is declared. (See test C52104Y.)

A null array with one dimension of length greater than `INTEGER'LAST` may raise `NUMERIC_ERROR` or `CONSTRAINT_ERROR` either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises `NUMERIC_ERROR` when the array type is declared. (See test E52103Y.)

In assigning one-dimensional array types, the expression appears to be evaluated in its entirety before `CONSTRAINT_ERROR` is raised when checking whether the expression's subtype is compatible with the target's subtype. In assigning two-dimensional array types, the expression does not appear to be evaluated in its entirety before `CONSTRAINT_ERROR` is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

Discriminated types.

During compilation, an implementation is allowed to either accept or reject an incomplete type with discriminants that is used in an access type definition with a compatible discriminant constraint. This implementation accepts such subtype indications. (See test E38104A.)

In assigning record types with discriminants, the expression appears to be evaluated in its entirety before `CONSTRAINT_ERROR` is raised when checking whether the expression's subtype is

compatible with the target's subtype. (See test C52013A.)

Aggregates.

In the evaluation of a multi-dimensional aggregate, all choices appear to be evaluated before checking against the index type. (See tests C43207A and C43207B.)

In the evaluation of an aggregate containing subaggregates, all choices are evaluated before being checked for identical bounds. (See test E43212B.)

All choices are evaluated before CONSTRAINT_ERROR is raised if a bound in a nonnull range of a nonnull aggregate does not belong to an index subtype. (See test E43211B.)

Representation clauses.

An implementation might legitimately place restrictions on representation clauses used by some of the tests. If a representation clause is used by a test in a way that violates a restriction, then the implementation must reject it.

Enumeration representation clauses containing noncontiguous values for enumeration types other than character and boolean types are supported. (See tests C35502I..J, C35502M..N, and A39005F.)

Enumeration representation clauses containing noncontiguous values for character types are supported. (See tests C35507I..J, C35507M..N, and C55B16A.)

Enumeration representation clauses for boolean types containing representational values other than (FALSE => 0, TRUE => 1) are supported. (See tests C35508I..J and C35508M..N.)

Length clauses with SIZE specifications for enumeration types are supported. (See test A39005B.)

Length clauses with STORAGE_SIZE specifications for access types are supported. (See tests A39005C and C87B62B.)

Length clauses with STORAGE_SIZE specifications for task types are supported. (See tests A39005D and C87B62D.)

Length clauses with SMALL specifications are supported. (See tests A39005E and C87B62C.)

There are restrictions for alignment clauses within record representation clauses. (See test A39005G.)

CONFIGURATION INFORMATION

Length clauses with SIZE specifications for derived integer types are supported. (See test C87B62A.)

Pragmas.

The pragma INLINE is not supported for procedures. The pragma INLINE is not supported for functions. (See tests LA3004A, LA3004B, EA3004C, EA3004D, CA3004E, and CA3004F.)

Input/output.

The package SEQUENTIAL_IO can be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101C, EE2201D, and EE2201E.)

The package DIRECT_IO can be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101H, EE2401D, and EE2401G.)

Modes IN_FILE and OUT_FILE are supported for SEQUENTIAL_IO. (See tests CE2102D and CE2102E.)

Modes IN_FILE, OUT_FILE, and INOUT_FILE are supported for DIRECT_IO. (See tests CE2102F, CE2102I, and CE2102J.)

RESET and DELETE are supported for SEQUENTIAL_IO and DIRECT_IO. (See tests CE2102G and CE2102K.)

Dynamic creation and deletion of files are supported for SEQUENTIAL_IO and DIRECT_IO. (See tests CE2106A and CE2106B.)

Overwriting to a sequential file does not truncate the file. (See test CE2208B.)

An existing text file can be opened in OUT_FILE mode, cannot be created in OUT_FILE mode, and cannot be created in IN_FILE mode. (See test EE3102C.)

More than one internal file can be associated with each external file for text I/O for both reading and writing. (See tests CE3111A..E (5 tests), CE3114B, and CE3115A.)

More than one internal file can be associated with each non-temporary external file for sequential I/O for both reading and writing. (See tests CE2107A..D (4 tests), CE2110B, and CE2111D.)

More than one internal file can be associated with each non-temporary external file for direct I/O for both reading and writing. (See tests CE2107F..I (5 tests), CE2110B, and CE2111H.)

CONFIGURATION INFORMATION

An internal sequential access file and an internal direct access file can be associated with a single external file for writing. (See test CE2107E.)

An external file associated with more than one internal file cannot be deleted but closed for SEQUENTIAL_IO, DIRECT_IO, and TEXT_IO. (See test CE2110B.)

Temporary sequential files are not given names. Temporary direct files are not given names. (See tests CE2108A and CE2108C.)

. Generics.

Generic subprogram declarations and bodies can be compiled in separate compilations. (See tests CA1012A and CA2009F.)

Generic .package declarations and bodies can be compiled in separate compilations. (See tests CA2009C, BC3204C, and BC3205D.)

Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.9 of the ACVC comprises 3122 tests. When this compiler was tested, 24 tests had been withdrawn because of test errors. The AVF determined that 213 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 159 executable tests that use floating-point precision exceeding that supported by the implementation. Modifications to the code, processing, or grading for 13 tests were required to successfully demonstrate the test objective. (See section 3.6.)

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
-----	_A_	_B_	_C_	_D_	_E_	_L_	-----
Passed	109	1048	1651	17	16	44	2885
Inapplicable	1	3	205	0	2	2	213
Withdrawn	3	2	18	0	1	0	24
TOTAL	113	1053	1874	17	19	46	3122

TEST INFORMATION

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER														TOTAL
	__2__	__3__	__4__	__5__	__6__	__7__	__8__	__9__	__10__	__11__	__12__	__13__	__14__	_____	
Passed	182	516	564	245	166	98	141	326	131	36	234	3	243	2885	
Inapplicable	22	57	111	3	0	0	2	1	6	0	0	0	11	213	
Withdrawn	2	13	2	0	0	1	2	0	0	0	2	1	1	24	
TOTAL	206	586	677	248	166	99	145	327	137	36	236	4	255	3122	

3.4 WITHDRAWN TESTS

The following 24 tests were withdrawn from ACVC Version 1.9 at the time of this validation:

B28003A	E28005C	C34004A	C35502P	A35902C
C35904A	C35A03E	C35A03R	C37213H	C37213J
C37215C	C37215E	C37215G	C37215H	C38102C
C41402A	C45614C	A74106C	C85018B	C87B04B
CC1311B	BC3105A	AD1A01A	CE2401H	

See Appendix D for the reason that each of these tests was withdrawn.

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. The applicability of a test to an implementation is considered each time a validation is attempted. A test that is inapplicable for one validation attempt is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 213 tests were inapplicable for the reasons indicated:

- . Tests C24113D..N (11 tests) contain lines of lengths greater than 80 characters which is not supported by this compiler.
- . A39005G uses an alignment clause with an alignment of 8 within a record representation clause.

- The following tests use LONG_INTEGER, which is not supported by this compiler:

C45231C	C45304C	C45502C	C45503C	C45504C
C45504F	C45611C	C45613C	C45631C	C45632C
B52004D	C55B07A	B55B09C		

- C45231D requires a macro substitution for any predefined numeric types other than INTEGER, SHORT_INTEGER, LONG_INTEGER, FLOAT, SHORT_FLOAT, and LONG_FLOAT. This compiler does not support any such types.
- C45531M, C45531N, C45532M, and C45532N use fine 48-bit fixed-point base types which are not supported by this compiler.
- C45531O, C45531P, C45532O, and C45532P use coarse 48-bit fixed-point base types which are not supported by this compiler.
- B86001D requires a predefined numeric type other than those defined by the Ada language in package STANDARD. There is no such type for this implementation.
- C86001F redefines package SYSTEM, but TEXT_IO is made obsolete by this new definition in this implementation and the test cannot be executed since the package REPORT is dependent on the package TEXT_IO.
- C96005B requires the range of type DURATION to be different from those of its base type; in this implementation they are the same.
- CA3004E, EA3004C, and LA3004A use the INLINE pragma for procedures, which is not supported by this compiler.
- CA3004F, EA3004D, and LA3004B use the INLINE pragma for functions, which is not supported by this compiler.
- CE2108A, CE2108C, CE3112A are inapplicable because temporary files (sequential, direct, text) do not have names.
- CE2107C, CE2107D, CE2107H, and CE2107I are inapplicable because multiple internal files cannot be associated with the same temporary external file. The proper exception is raised when multiple access is attempted.
- Tests CE2110B and CE3114B contain attempts to DELETE one of two files that access the same external file. In this implementation, such an attempt closes the internal file, but fails to delete the external file, raising USE_ERROR. The tests' subsequent attempt to again DELETE the internal file raises STATUS_ERROR, since that file is no longer open. The AVO accepts this behavior while the issue is reviewed further.

TEST INFORMATION

- . Test CE3111B assumes that a PUT - operation writes data to an external file immediately. For this implementation data are written to a buffer first, thus this test's attempt to immediately GET data raises END_ERROR. The AVO ruled that this behavior is acceptable.
- . Test CE3202A requires that the name function returns strings which identify the standard input and output files. The underlying operating system does not support this requirement. The AVO ruled that this behavior is acceptable.
- . The following 159 tests require a floating-point accuracy that exceeds the maximum of 18 digits supported by this implementation:

C241130..Y (11 tests)	C357050..Y (11 tests)
C357060..Y (11 tests)	C357070..Y (11 tests)
C357080..Y (11 tests)	C358020..Z (12 tests)
C452410..Y (11 tests)	C453210..Y (11 tests)
C454210..Y (11 tests)	C455210..Z (12 tests)
C455240..Z (12 tests)	C456210..Z (12 tests)
C456410..Y (11 tests)	C460120..Z (12 tests)

3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS

It is expected that some tests will require modifications of code, processing, or evaluation in order to compensate for legitimate implementation behavior. Modifications are made by the AVF in cases where legitimate implementation behavior prevents the successful completion of an (otherwise) applicable test. Examples of such modifications include: adding a length clause to alter the default size of a collection; splitting a Class B test into subtests so that all errors are detected; and confirming that messages produced by an executable test demonstrate conforming behavior that wasn't anticipated by the test (such as raising one exception instead of another).

Modifications were required for 13 Class B tests.

The following Class B tests were split because syntax errors at one point resulted in the compiler not detecting other errors in the test:

B22003A	B24009A	B29001A	B38003A	B38009A
B38009B	B51001A	B91001H	BC2001D	BC2001E
BC3204B	BC3205B	BC3205D		

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.9 produced by the SYSTEAM Ada Cross Compiler VAX/VMS x MC68020/OS-9 was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behavior on all inapplicable tests.

3.7.2 Test Method

Testing of the SYSTEAM Ada Cross Compiler VAX/VMS x MC68020/OS-9 using ACVC Version 1.9 was conducted on-site by a validation team from the AVF. The configuration consisted of a VAX 8530 host operating under VMS, Version 4.6, and a KWS EB 68/20 CPU3 target operating under OS-9, Version 2.0. The host and target computers were linked via V24 connection.

A magnetic tape containing all tests except for withdrawn tests and tests requiring unsupported floating-point precisions was taken on-site by the validation team for processing. Tests that make use of implementation-specific values were customized before being written to the magnetic tape. Tests requiring modifications during the prevalidation testing were included in their modified form on the magnetic tape.

The contents of the magnetic tape were loaded directly onto the host computer.

After the test files were loaded to disk, the full set of tests was compiled on the VAX 8530. Object files were linked and executed on the target. Results were printed from the host computer, with results being transferred to the host computer via V24 connection.

The compiler was tested using command scripts provided by SYSTEAM KG and reviewed by the validation team. The compiler was tested using all default settings.

Tests were compiled, linked, and executed (as appropriate) using a single host computer and a single target computer. Test output, compilation listings of Class B tests and tests that raised an error during compilation, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

3.7.3 Test Site

Testing was conducted at SYSTEAM KG at Karlsruhe and was completed on 88-02-29.

APPENDIX A

DECLARATION OF CONFORMANCE

SYSTEM KG has submitted the following Declaration of
Conformance concerning the SYSTEM Ada Cross Compiler
VAX/VMS x MC68020/OS-9.

DECLARATION OF CONFORMANCE

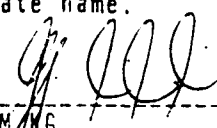
Compiler Implementor: SYSTEAM KG
Ada Validation Facility: IABG m.b.H., Dept SZT
Ada Compiler Validation Capability (ACVC) Version: 1.9

Base Configuration

Base Compiler Name: SYSTEAM Ada Cross Compiler VAX/VMS x MC68020/OS-9
Base Compiler Version: Version 1.61
Host Architecture ISA: VAX 8530 VMS 4.6
Target Architecture ISA: KWS EB 68/20 CPU3 OS-9 2.0

Implementor's Declaration

I, the undersigned, representing SYSTEAM KG, have implemented no deliberate extensions to the Ada Language Standard ANSI/MIL-STD-1815A in the compiler(s) listed in this declaration. I declare that SYSTEAM KG is the owner of record of the Ada language compiler(s) listed above and, as such, is responsible for maintaining said compiler(s) in conformance to ANSI/MIL-STD-1815A. All certificates and registrations for Ada language compiler(s) listed in this declaration shall be made only in the owner's corporate name.

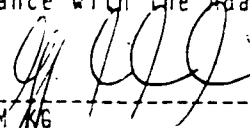


SYSTEAM KG
Dr. Winterstein,

Date: March 15, 1988

Owner's Declaration

I, the undersigned, representing SYSTEAM KG, take full responsibility for implementation and maintenance of the Ada compiler(s) listed above, and agree to the public disclosure of the final Validation Summary Report. I further agree to continue to comply with the Ada trademark policy, as defined by the Ada Joint Program Office. I declare that all of the Ada language compilers listed, and their host/target performance, are in compliance with the Ada Language Standard ANSI/MIL-STD-1815A.



SYSTEAM KG
Dr. Winterstein,

Date: March 15, 1988

APPENDIX B

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of the SYSTEAM Ada Cross Compiler VAX/VMS x MC68020/OS-9, Version 1.61, are described in the following sections, which discuss topics in Appendix F of the Ada Standard. They are taken from the vendor's User Manual. Implementation-specific portions of the package STANDARD are defined as follows.

PACKAGE standard IS

TYPE boolean IS (false, true);

TYPE short_integer IS RANGE - 32_768 .. 32_767;

TYPE integer IS RANGE - 2_147_483_648 .. 2_147_483_647;

TYPE short_float IS DIGITS 6 RANGE - 16#0.FFFFFFF#E32 .. 16#0.FFFFFFF#E32;

TYPE float IS DIGITS 15 RANGE
- 16#0.FFFFFFFFFFFFF8#E256 .. 16#0.FFFFFFFFFFFFF8#E256;

TYPE long_float IS DIGITS 18 RANGE
- 16#0.FFFFFFFFFFFFFFFFFF#E4096 .. 16#0.FFFFFFFFFFFFFFFFFF#E4096;

-- TYPE character IS ... as in [ADA,Appendix C]

-- FOR character USE ... as in [ADA,Appendix C]

-- PACKAGE ascii IS ... as in [ADA,Appendix C]

-- Predefined subtypes and string types ... as in [ADA,Appendix C]

TYPE duration IS DELTA 2#1.0#E-14 RANGE
- 131_072.0 .. 131_071.999_938_964_843_75;

-- The predefined exceptions are as in [ADA,Appendix C]

END standard;

13 Representation clauses and implementation-dependent features

In this chapter we follow the section numbering of Chapter 13 of [ADA] and provide notes for the use of the features described in each section.

13.1 Representation clauses

Pragma PACK : as stipulated in [ADA,§13.1], this pragma may be given for a record or array type. It causes the cross compiler to select a representation for this type such that gaps between the storage areas allocated to consecutive components are minimized, but it does not affect the mapping of each component onto storage. An even greater saving in storage space can be achieved by using the implementation-defined pragma **SQUEEZE** (see below).

Pragma SQUEEZE : this is an implementation-defined pragma which takes the same argument as the predefined language pragma **PACK** and is allowed at the same positions. It causes the cross compiler to select a representation for the argument type that needs minimal storage space. By contrast, pragma **PACK** (see above) only leads to representations which cause components of objects of its argument-types to start on storage-unit-bounds.

13.2 Length clauses

SIZE

- for all integer, fixed point and enumeration types the value must be ≤ 32 ;
- for **SHORT_FLOAT** types the value must be $= 32$ (this is the amount of storage which is associated with these types anyway);
- for **FLOAT** types the value must be $= 64$ (this is the amount of storage which is associated with these types anyway).
- for access types the value must be $= 32$ (this is the amount of storage which is associated with these types anyway);

The implementation does support size specification for record and array types, but if a size specification is given for type which is a derived type of a record or array type, then the value of the expression in the size specification must be equal to the number of bits to be allocated to objects of the parent type (in other words, size specification for records or arrays cannot be used to make 'SIZE of a derived type differ from that of its parent type).

The implementation does not support size specification for task types. If any of the above restrictions are violated, the cross compiler responds with a **RESTRICTION** error message in the cross compiler listing.

STORAGE.SIZE

Collection size: If no length clause is given, the storage space needed to contain objects designated by values of the access type and by values of other types derived from it is extended dynamically at runtime as needed. If, on the other hand, a length clause is given, the number of storage units stipulated in the length clause is reserved, and no dynamic extension at runtime occurs.

Storage for Tasks: The memory space reserved for a task is 4K bytes if no length clause is given. If the task is to be allotted either more or less space, a length clause must be given for its task type, and then all tasks of this type will be allotted the amount of space stipulated in the length clause (the activation of a small task requires about 1.4K bytes). Whether a length clause is given or not, the space allotted is not extended dynamically at runtime.

SMALL

the value given in a specification of small for a fixed point type must be a power of two. If this restriction is violated, the cross compiler responds with a **RESTRICTION** error message in the compiler listing.

13.3 Enumeration representation clauses

The implementation places no restrictions on enumeration representation clauses.

13.4 Record representation clauses

The value of the expression given in an alignment clause must be 0, 1, 2 or 4. If this restriction is violated, the cross compiler responds with a **RESTRICTION** error message in the compiler listing.

The implementation places no restrictions on component clauses other than those in the language ([ADA,§13.4(6,7)]).

There are no implementation-generated names denoting implementation-dependent components (cf. [ADA,§13.4(8)]).

13.5 Address clauses

Address clauses are supported only for objects declared by an object declaration whose type is not a task type. If an address clause is given for a subprogram, package, task unit or single entry, the cross compiler responds with a `RESTRICTION` error message in the compiler listing.

Since the type `ADDRESS` in the package `SYSTEM` is declared as a private type, the `simple_expression` in any address clause must be a function call of one of the functions `convert_address` declared in the package `SYSTEM`.

Example:

```
FOR my_record USE AT system.convert_address("01234EFO");
```

When an address is given as a hexadecimal string as in this example, a string of length 8 should be given (if a shorter string is given, the cross compiler adjoins an appropriate number of '0' to the right-hand end of the string; and giving a longer string causes `CONSTRAINT_ERROR` to be raised at run-time when `convert_address` is called).

An object for which an address clause is given must not require initialization (whether explicit or implicit); if it does, the program is erroneous, the cross compiler issues a warning message and the effect at runtime is unpredictable. It follows from this that address clauses must not be given for objects whose type has a discriminant part (cf. [ADA,§3.7.2(8)]) or for objects whose type is an access type (cf. [ADA,§3.2.1(10)]), because these always require initialization.

13.6 Change of representation

The implementation places no restrictions on changes of representation, except that, if a size specification is given for a record type which is a derived type, then the value of the expression in the size specification must be equal to the number of bits to be allocated to objects of the parent type (in other words, size specification for records cannot be used to make `'SIZE` of a derived record type differ from that of its parent type).

13.7 The package SYSTEM

See §F.3.

The pragmas `SYSTEM_NAME`, `STORAGE_UNIT` and `MEMORY_SIZE` have no effect.

13.7.1 System-dependent named numbers

See Chapter 15, §F.3.

13.7.2 Representation attributes

These are all implemented.

13.7.3 Representation attributes of real types

These are all implemented.

13.8 Machine code insertions

A package `MACHINE_CODE` is not provided and machine code insertions are not supported.

13.9 Interface to other languages

This is provided for assembly language subprograms. For each Ada subprogram for which

```
PRAGMA interface (assembler, <Ada_Name>);
```

is specified, an assembly language program implementing the body of the Ada subprogram must be provided.

Some conventions must be obeyed when writing the assembly source. To this end four macros (PARAMS, LOCALS, ROUTINE and RETURN) are provided for use with the R68020. The source file must be written in the following form:

```

PSECT  psect_name.0.0.0.0.0
USE    /ada/external.defs          (1)
*
PARAMS size_of_paramterblock      (2)
p1 EQU  offset1                    (3)
p2 EQU  offset2
...
LOCALS                             (4)
11 LO.W  1                          (5)
12 LO.L  2
...
entry_label: ROUTINE               (6)
MOVE.W  (p1,A4),D0                  (7)
CLR.L   (12,A5)                     (8)
...
RETURN                             (9)
*
PARAMS
...
*
ENDS
```

Within one psect the bodies for more than one subprogram may be given. The body of one subprogram starts with a call of macro PARAMS (2) and ends with the next call of marco RETURN (9).

The parameter definitions are started by the call of the macro PARAMS (2). The size of the parameter block must be given as argument. The names of the parameters

are introduced by EQU-directives (3). The offsets of the parameters and the size of the parameter block are determined by the compiler when compiling the subprogram specification but must be included here by the user. This is a source of inconsistencies. Therefore it is recommended to have just one record parameter. In this case, the address of the parameter is passed (i.e. size of parameter block is 4 and the offset of the only parameter is 0). The layout of the record can be controlled in the Ada source by a representation clause. Another way to get the correct offsets is to compile a call of the external subprogram with option SYMBOLIC_CODE and look at the code of the call.

The call of macro PARAMS must be present even if the external subprogram does not have any parameters.

In the code of the external subprogram the actual parameters are accessed in the form (parameter_name,A4) (7).

The call of the macro LOCALS (4) introduces the definitions of local variables. These variables are allocated on the runtime stack of the Ada program. Each variable is defined using the LO-directive (5).

The call of macro LOCALS must be present even if the external subprogram does not have any local variables.

In the code of the external subprogram, the local variables are accessed in the form (variable_name,A5) (8).

After the call of the macro ROUTINE (6) the code of the external subprogram follows. It is terminated by the next call of the macro RETURN (9), which completes the subprogram and performs the RTS-instruction. Immediately before the RETURN is executed, the registers A4, A5 and A6 must have the same contents as at the beginning of the external subprogram. All others registers may have values different to those they had on entry.

The entry point of the external subprogram is indicated by the label on the line containing the call of the macro ROUTINE (6). As entry label the Ada subprogram name truncated to 8 characters must be used. It must be followed by a colon.

The macro definitions for PARAMS, LOCALS, ROUTINE and RETURN are contained in the file /ada/external.defs. It must be included with a USE-directive (1).

Process the assembly source with the R68020 by calling the command file

```
OS9$ make -f=/ada/external source=<source file> -  
        list=<listing file> -  
        name=<name> -  
        [library=<directory>]
```

<source file> gives the name of the source file. <listing file> names the file containing the assembly listing. The object module generated is stored in the target library on file <directory>/EXT.<name>. Therefore <name> must be determined so that no name clashes occur with the object files of other external subprograms. All object files <directory>/EXT.* are included in an executable program during linking if required. To this end they are merged into the object module library <directory>/EXTERNAL.OML which is specified as library (i.e. -l=<directory>/EXTERNAL.OML) in the call of the L68.

The default for <directory> is ALB.

13.10 Unchecked programming

13.10.1 Unchecked storage deallocation

The implementation does not support unchecked storage deallocation. (The generic procedure UNCHECKED_DEALLOCATION is provided, but the only effect of calling an instantiation of this procedure with an object X as actual parameter is

```
X := NULL;
```

i.e. no storage is reclaimed.)

However, the implementation does provide an implementation-defined package COLLECTION_MANAGER which fulfils a similar function (cf. Chapter 12).

13.10.2 Unchecked type conversions

The implementation does support unchecked type conversions. Note that if
target_type'size > source_type'size,

the result value of the unchecked conversion is unpredictable.

14 Input-output

In this chapter we follow the section numbering of Chapter 14 of [ADA] and provide notes for the use of the features described in each section.

14.1 External files and file objects

The total number of open text files (including the two standard files), sequential files and direct files must not exceed 10 for each class. Any attempt to exceed this limit raises the exception `USE_ERROR`.

File sharing is allowed for reading and writing without any restriction.

The following restrictions apply to the generic actual parameter for `ELEMENT_TYPE`:

- input/output of access types is not defined.
- input/output is not possible for an object whose (sub)type has a size which is not a multiple of `SYSTEM.STORAGE_UNIT`. Such objects may only exist for types for which a representation clause or pragma `SQUEEZE` is given. `USE_ERROR` will be raised by any attempt to read or write such an object or to open or create a file for such a (sub)type.

14.2 Sequential and direct files

Sequential and direct files are represented by OS9 RBF files with fixed-length or variable-length records. Each element of the file is stored in one record.

14.2.1 File management

Since there is a lot to say about this section, we shall introduce subsection numbers which do not exist in [ADA].

14.2.1.1 The NAME and FORM parameters

The NAME parameter string must be an OS9 file name. The function NAME will return a file name string which is the file name of the file opened or created.

The Syntax of the FORM parameter string is defined by:

```
form_parameter ::= [ form_specification { . form_specification } ]  
form_specification ::= keyword [ => value ]  
keyword ::= identifier  
value ::= identifier | string_literal | numeric_literal
```

For identifier, numeric_literal, string_literal see [ADA,Appendix E]. Only an integer literal is allowed as numeric_literal (see [ADA,§2.4]).

In the following, the form specifications which are allowed for all files are described.

```
ALLOCATION => numeric_literal
```

This value specifies the number of blocks which are allocated initially; it is only used in a create operation and ignored in an open operation. The default value for the initial file size is 0.

```
RECORD_SIZE => numeric_literal
```

This value specifies the record size in bytes. This form specification is only allowed for files with fixed record format. If the value is specified for an existing file, it must agree with the value of the external file.

By default, ELEMENT_TYPE'SIZE / SYSTEM.STORAGE_UNIT will be chosen as record size if the evaluation of this expression does not raise an exception. Otherwise 256 is used as default. In this case, the attempt to write or read a larger record will raise USE_ERROR.

If a fixed record format is used, all objects written to a file which are shorter than the record size are filled up with zeros (ASCII.NUL). An attempt to write an element

which is larger than the specified record size will result in the exception `USE_ERROR` being raised. This can only occur if the record size is specified explicitly or if the evaluation of the expression `ELEMENT_TYPE'SIZE / SYSTEM.STORAGE_UNIT` raises an exception.

14.2.1.2 Sequential files

A sequential file is represented by an RBF file with either fixed-length or variable-length records which may be specified by the `form` parameter.

If a fixed record format is used, all objects written to a file which are shorter than the maximum record size are filled up with zeros (`ASCII.NUL`).

`RECORD_FORMAT => VARIABLE | FIXED`

This form specification is used to specify the record format. If the format is specified for an existing file, it must agree with the format of the external file.

Fixed record size is used as default. It means that every record is written with the size specified as record size.

Variable record size means that each record is written with its actual length. A read operation transfers as many bytes as are allocated for the receiving object, i.e. the object supplied as out-parameter to the read procedure. Since the record length is not stored on the external file, care has to be taken if the file is read again.

14.2.1.3 Direct files

The implementation dependent type `COUNT` defined in the package specification of `DIRECTIO` has an upper bound of :

`COUNT'LAST = 2.147.483.647 (= INTEGER'LAST)`

Direct files are represented by OS9 RBF files with fixed-length records.

14.3 Text input/output

Additionally to the packages defined in Chapter 14 of the LRM, the package `MINI_IO` is provided. It provides only a procedure `PUT_LINE`, which can be used to write a single line onto standard output. Its Ada specification is given in §14.8. It provides an alternative to `TEXT_IO.PUT_LINE` in applications which do not require the full power of `TEXT_IO` and do not want to include a lot of code which is not required.

Text files are represented as RBF or SCF files depending on whether the file name denotes a disk file or a terminal device. Each line consists of a sequence of characters terminated by an `ASCII.CR`.

A page terminator is represented as a line consisting of a single `ASCII.FF`. A page terminator is always preceded by a line terminator (i.e. `ASCII.CR`).

A file terminator is not represented explicitly in the external file; the end of the file is taken as a file terminator. A page terminator is assumed to precede the end of the file if there is not explicitly one as the last record of the file.

14.3.1 File management

In the following, the form specifications which are only allowed for text files or have a special meaning for text files are described.

`CHARACTER_IO`

The predefined package `TEXT_IO` was designed for sequential text files; moreover, this implementation always uses sequential files with a record structure, even for terminal devices. It therefore offers no language-defined facilities for modifying data previously written to the terminal (e.g. changing characters in a text which is already on the terminal screen) or for outputting characters to the terminal without following them by a line terminator. It also has no language-defined provision for input of single characters from the terminal (as opposed to lines, which must end with a line terminator, so that in order to input one character the user must type in that character and then a line terminator) or for suppressing the echo on the terminal of characters typed in at the keyboard.

For these reasons, in addition to the input/output facilities with record structured external files, another form of input/output is provided for text files: It is possible to

transfer single characters from/to a terminal device. This form of input/output is specified by the keyword `CHARACTER_IO` in the form string. If character i/o is specified, no other form specification is allowed and the file name must denote a terminal device.

For an infile, the external file (associated with a terminal) is considered to contain a single line. Arbitrary characters (including all control characters) may be read; a character read is not echoed to the terminal.

For an outfile, arbitrary characters (including all control characters and escape sequences) may be written on the external file (terminal). A line terminator is represented as `ASCII.CR` followed by `ASCII.LF`, a page terminator is represented as `ASCII.FF` and a file terminator is not represented on the external file.

14.3.2 Default input and output files

The Ada standard input and output files are associated with the corresponding standard files in OS9.

14.3.10 Implementation-defined types

The implementation dependent types `COUNT` and `FIELD` defined in the package specification of `TEXT_IO` have the following upper bounds :

```
COUNT'LAST = 2_147_483_647 (= INTEGER'LAST)
```

```
FIELD'LAST = 255
```

14.4 Exceptions in input-output

For each of `NAME_ERROR`, `USE_ERROR`, `DEVICE_ERROR` and `DATA_ERROR` we list the conditions under which that exception can be raised. The conditions under which the other exceptions declared in the package `IO_EXCEPTIONS` can be raised are as described in [ADA,§14.4].

`NAME_ERROR`

- in an `OPEN` operation, if the specified file does not exist;
- in a `CREATE` operation, if the specified file already exists;
- if the name parameter in a call of the `CREATE` or `OPEN` procedure is not a legal OS9 file specification string; for example, if it contains illegal characters, is too long or is syntactically incorrect; and also if it contains wild cards, even if that would specify a unique file.

`USE_ERROR`

- if an attempt is made to increase the total number of open files (including the two standard files) so that there are more than 10 in one of the three file classes text, sequential and direct;
- whenever an error occurred during an operation of the underlying OS9 system. This may happen if an internal error was detected, an operation is not possible for reasons depending on the file or device characteristics, a size restriction is violated, a capacity limit is exceeded or for similar reasons; in general it is only guaranteed that a file which is created by an Ada program may be reopened and read successfully by another program if the file types and the form strings are the same;
- if the function `NAME` is applied to a temporary file;
- if a given `FORM` parameter string does not have the correct syntax or if a condition on an individual form specification described in §§14.2-3 is not fulfilled;
- if an attempt is made to open or create a sequential or direct file for an element type whose size is not a multiple of `SYSTEM.STORAGE_UNIT`; or if an attempt is made to read or write an object whose (sub)type has a size which is not a multiple of `SYSTEM.STORAGE_UNIT` (such situations can only arise for types for which a representation clause or pragma `SQUEEZE` is given);
- if an attempt is made to write or read to/from a file with fixed record format a record which is larger than the record size laid down when the file was opened (cf. §14.2.1.1);

`DEVICE_ERROR`

is never raised. Instead of this exception the exception `USE_ERROR` is raised whenever an error occurred during an operation of the underlying OS9 system.

DATA_ERROR

- the conditions under which DATA_ERROR is raised in the package TEXT_IO are laid down in [ADA]; the following notes apply to the packages SEQUENTIAL_IO and DIRECT_IO:
- by the procedure READ if the size of a variable-length record in the external file to be read exceeds the storage size of the given variable or else the size of a fixed-length record in the external file to be read exceeds the storage size of the given variable which has exactly the size ELEMENT_TYPE'size.
- In general, the exception DATA_ERROR is not raised by the procedure READ if the element read is not a legal value of the element type.
- by the procedure READ if an element with the specified position in a direct file does not exist; this is only possible if the file is associated with a relative or an indexed file.

14.6 Low level input-output

We give here the specification of the package LOW_LEVEL_IO:

```
PACKAGE low_level_io IS

  TYPE device_type IS (null_device);

  TYPE data_type IS
    RECORD
      NULL;
    END RECORD;

  PROCEDURE send_control    (device : device_type;
                             data   : IN OUT data_type);

  PROCEDURE receive_control (device : device_type;
                             data   : IN OUT data_type);

END low_level_io;
```

Note that the enumeration type DEVICE_TYPE has only one enumeration value, NULL_DEVICE; thus the procedures SEND_CONTROL and RECEIVE_CONTROL can be called, but SEND_CONTROL will have no effect on any physical device and the value of the actual parameter DATA after a call of RECEIVE_CONTROL will have no physical significance.

14.8 Specification of the package MINI_IO

PACKAGE mini_io IS

 PROCEDURE put_line (item : string);

END mini_io;

15 Appendix F

This is the Appendix F required in [ADA], in which all implementation-dependent characteristics of an Ada implementation are described.

F.1 Implementation-dependent pragmas

The form, allowed places, and effect of every implementation-dependent pragma is stated in this section.

F.1.1 Predefined language pragmas

The form and allowed places of the following pragmas are defined by the language; their effect is (at least partly) implementation-dependent and stated here. All the other pragmas listed in Appendix B of [ADA] are implemented and have the effect described there.

CONTROLLED

has no effect.

INLINE

has no effect; inline inclusion is never done.

INTERFACE

is implemented for Assembly language; see §13.9 of this manual for details.

MEMORY_SIZE

has no effect.

OPTIMIZE

has no effect.

PACK

see §13.1.

PRIORITY

There are two implementation-defined aspects of this pragma: First, the range of the subtype PRIORITY, and second, the effect on scheduling of not giving this pragma for a task or main program. The range of subtype PRIORITY is 0 .. 255, as declared in the predefined library package SYSTEM (see §F.3); and the effect on scheduling of leaving the priority of a task or main program undefined by not giving pragma PRIORITY for it is the same as if pragma PRIORITY 0 had been given (i.e. the task has the lowest priority). Moreover, in this implementation the

package SYSTEM must be named by a with clause of a compilation unit if the predefined pragma PRIORITY is used within that unit.

SHARED

has no effect. Note, however, that the implementation of tasking is such that every variable is treated as if pragma SHARED had been given for it.

STORAGE_UNIT

has no effect.

SUPPRESS

has no effect, but see §F.1.2 for the implementation-defined pragma SUPPRESS_ALL.

SYSTEM_NAME

has no effect.

*F.1.2 Implementation-defined pragmas***SQUEEZE**

see §13.1.

SUPPRESS_ALL

causes all the run-time checks described in [ADA, §11.7] except ELABORATION_CHECK to be suppressed; this pragma is only allowed at the start of a compilation before the first compilation unit; it applies to the whole compilation.

F.2 Implementation-dependent attributes

The name, type and implementation-dependent aspects of every implementation-dependent attribute is stated in this chapter.

F.2.1 Language-defined attributes

The name and type of all the language-defined attributes are as given in [ADA]. We note here only the implementation-dependent aspects.

STORAGE_SIZE

The value delivered by this attribute applied to an access type is as follows:

If a length specification (**STORAGE_SIZE**, see §13.2) has been given for that type (static collection), the attribute delivers that specified value.

In case of a dynamic collection, i.e. no length specification by **STORAGE_SIZE** given for the access type, the attribute delivers the number of storage units currently allocated for the collection. Note that dynamic collections are extended if needed.

If the collection manager (cf. Chapter 12) is used for a dynamic collection the attribute delivers the number of storage units currently allocated for the collection. Note that in this case the number of storage units currently allocated may be decreased by release operations.

F.2.2 Implementation-defined attributes

There are no implementation-defined attributes.

F.3 Specification of the package SYSTEM

The package SYSTEM of ([ADA,§13.7]) is reprinted here with all implementation-dependent characteristics and extensions filled in.

PACKAGE system IS

TYPE address IS PRIVATE;

TYPE name IS (motorola_68020);

system_name : CONSTANT name := motorola_68020;

storage_unit : CONSTANT := 8;

memory_size : CONSTANT := 2_147_483_648;

min_int : CONSTANT := - 2_147_483_648;

max_int : CONSTANT := 2_147_483_647;

max_digits : CONSTANT := 18;

max_mantissa : CONSTANT := 31;

fine_delta : CONSTANT := 2#1.0#E-30;

tick : CONSTANT := 0.2E-6;

SUBTYPE priority IS integer RANGE 0 .. 255;

TYPE universal_integer IS RANGE min_int .. max_int;

SUBTYPE external_address IS string;

SUBTYPE byte IS integer RANGE 0..255;

TYPE long_word IS ARRAY (0..3) OF byte;

PRAGMA PACK (long_word);

FUNCTION convert_address (addr : external_address) RETURN address;

FUNCTION convert_address (addr : address) RETURN external_address;

FUNCTION convert_address (addr : long_word) RETURN address;

FUNCTION convert_address (addr : address) RETURN long_word;

FUNCTION "+" (addr : address; offset : integer) RETURN address;

PRIVATE

-- private declarations

END system;

F.4 Restrictions on representation clauses

See §§13.2-13.5 of this manual.

F.5 Conventions for implementation-generated names

There are no implementation-generated names denoting implementation-dependent components ([ADA,§13.4]).

F.6 Expressions in address clauses

Address clauses ([ADA,§13.5]) are supported only for objects. The object starts at the given address.

F.7 Restrictions on unchecked conversions

See §13.10.2 of this manual.

F.8 Characteristics of the input-output packages

The implementation-dependent characteristics of the input-output packages as defined in Chapter 14 of [ADA] are reported in Chapter 14 of this manual.

APPENDIX C
TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below.

<u>Name and Meaning</u> -----	<u>Value</u> -----
\$BIG_ID1 Identifier the size of the maximum input line length with varying last character.	(1..79 => 'A', 80 => '1')
\$BIG_ID2 Identifier the size of the maximum input line length with varying last character.	(1..79 => 'A', 80 => '2')
\$BIG_ID3 Identifier the size of the maximum input line length with varying middle character.	(1..40=>'A',41=>'3',42..80=>'A')
\$BIG_ID4 Identifier the size of the maximum input line length with varying middle character.	(1..40=>'A',41=>'4',42..80=>'A')
\$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	(1..77=>'0')&"298"

TEST PARAMETERS

Name_and_Meaning-----	Value-----
*BIG_REAL_LIT A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	(1..75=>'0')&"690.0"
*BIG_STRING1 A string literal which when catenated with BIG_STRING2 yields the image of BIG_ID1.	(1=>'"',2..41=>'A',42=>'")
*BIG_STRING2 A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1.	(1=>'"',2..40=>'A',41=>'1',42=>'")
*BLANKS A sequence of blanks twenty characters less than the size of the maximum line length.	(1..60 => ' ')
*COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	2147483647
*FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	255
*FILE_NAME_WITH_BAD_CHARS An external file name that either contains invalid characters or is too long.	abc!@def.dat
*FILE_NAME_WITH_WILD_CARD_CHAR An external file name that either contains a wild card character or is too long.	abc*def.dat
*GREATER_THAN_DURATION A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	0.0

<u>Name and Meaning</u> -----	<u>Value</u> -----
\$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	200_000.0
\$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	x\$'yz.dat
\$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	(1..40 => 'A')
\$INTEGER_FIRST A universal integer literal whose value is INTEGER'FIRST.	-2147483648
\$INTEGER_LAST A universal integer literal whose value is INTEGER'LAST.	2147483647
\$INTEGER_LAST_PLUS_1 A universal integer literal whose value is INTEGER'LAST + 1.	2147483648
\$LESS_THAN_DURATION A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	-0.0
\$LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-200_000.0
\$MAX_DIGITS Maximum digits supported for floating-point types.	18
\$MAX_IN_LEN Maximum input line length permitted by the implementation.	80
\$MAX_INT A universal integer literal whose value is SYSTEM.MAX_INT.	2147483647
\$MAX_INT_PLUS_1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	2147483648

TEST PARAMETERS

Name and Meaning-----	Value-----
\$MAX_LEN_INT_BASED_LITERAL A universal integer based literal whose value is 2#11# with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	"2:"&(3..77=>'0')&"11:"
\$MAX_LEN_REAL_BASED_LITERAL A universal real based literal whose value is 16:F.E: with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	"16:"&(4..76=>'0')&"F.E:"
\$MAX_STRING_LITERAL A string literal of size MAX_IN_LEN, including the quote characters.	(1=>'"',2..79=>'A',80 =>'")
\$MIN_INT A universal integer literal whose value is SYSTEM.MIN_INT.	-2147483648
\$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.	\$NAME
\$NEG_BASED_INT A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.	16#FFFFFFF#

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 24 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form "AI-ddddd" is to an Ada Commentary.

- . B28003A: A basic declaration (line 36) incorrectly follows a later declaration.
- . E28005C: This test requires that "PRAGMA LIST (ON);" not appear in a listing that has been suspended by a previous "PRAGMA LIST (OFF);". The Ada Standard is not clear on this point, and the matter will be reviewed by the AJPO.
- . C34004A: The expression in line 168 yields a value outside the range of the target type T, but there is no handler for CONSTRAINT_ERROR.
- . C35502P: The equality operators in line 62 and 69 should be inequality operators.
- . A35902C: The assignment in line 17 of the nominal upper bound of a fixed-point type to an object raises CONSTRAINT_ERROR, for that value lies outside of the actual range of the type.
- . C35904A: The elaboration of the fixed-point subtype on line 28 wrongly raises CONSTRAINT_ERROR, because its upper bound exceeds that of the type.
- . C35A03E and C35A03R: These tests assume that attribute 'MANTISSA returns 0 when applied to a fixed-point type with a null range, but the Ada Standard does not support this assumption.
- . C37213H: The subtype declaration of SCONS in line 100 is incorrectly expected to raise an exception when elaborated.

WITHDRAWN TESTS

- . C37213J: The aggregate in line 451 incorrectly raises CONSTRAINT_ERROR.
- . C37215C, C37215E, C37215G and C37215H: Various discriminant constraints are incorrectly expected to be incompatible with type CONS.
- . C38102C: The fixed-point conversion on line 23 wrongly raises CONSTRAINT_ERROR.
- . C41402A: The attribute 'STORAGE_SIZE is incorrectly applied to an object of an access type.
- . C45614C: The function call of IDENT_INT in line 15 uses an argument of the wrong type.
- . A74106C, C85018B, C67804B and CC1311B: A bound specified in a fixed-point subtype declaration lies outside of that calculated for the base type, raising CONSTRAINT_ERROR. Errors of this sort occur at lines 37 & 59, 142 & 143, 16 & 48, and 252 & 253 of the four tests, respectively.
- . BC3105A: Lines 159 through 168 expect error messages, but these lines are correct Ada.
- . AD1A01A: The declaration of subtype SINT3 raises CONSTRAINT_ERROR for implementations which select INT'SIZE to be 16 or greater.
- . CE2401H: The record aggregates in lines 105 and 117 contain the wrong values.